Zespół: **CZW/TP/11**

Michał Ryśkiewicz, 241383

Wrocław, dn. 13 maja 2020

Ocena:

Oddano:

Testowanie pierwszości liczb

Rok akad. 2019/2020, kierunek: INF

PROWADZĄCY:

dr hab. inż. Janusz Biernat

Spis treści

1 Wprowadzenie

Liczbami pierwszymi określamy liczby naturalne większe od 1, które dzielą się przez dokładnie dwie liczby: jedynkę i samą siebie. Ich dodatkową cechą jest też to, że są one ułożone losowo na osi liczbowej. Te własności próbowały udowodnić lub obalić dziesiątki tysięcy ludzi na przestrzeni wieków.

Jednym z pierwszych znanych nam matematyków, który pochylił się nad problemem znalezienia sposobu sprawdzenia czy dana liczba jest pierwsza był Erastotenes. Zaproponował on nieskomplikowany algorytm znany dzisiaj jako "Sito Erastotenesa", polegający na wybraniu najmniejszej liczby z przedziału [2, n] i wykreślaniu wszystkich jej wielokrotności, dopóki najmniejsza liczba nie będzie równa n. Inne bardziej wyrafinowane testy pierwszości zaczęły pojawiać się dopiero od ok. XVII w. n.e.

1.1 Założenia, ograniczenia i opis programu

W projekcie zostały zaimplementowane poniższe algorytmy:

- test pierwszości Fermata,
- test pierwszości Millera-Rabina w wersji probabilistycznej,
- test pierwszości Millera-Rabina w wersji deterministycznej,
- test pierwszości Agrawal-Kayal-Saxena (AKS).

Końcowy program został napisany w języku C++, a wszystkie algorytmy, za wyjątkiem AKS, zostały zaimplementowane w języku C++ i Python, a następnie zostały one poddane analizie porównawczej. Algorytm AKS został napisany jedynie w języku Python, a uruchamianie go w gotowym programie nie jest zalecane ze względu na generowanie błędów w momencie uruchomienia go poza powłoką Pythona.

Dodatkowo program umożliwia użytkownikowi wprowadzenie liczby dodatniej składającej się maksymalnie z 4094 cyfr oraz wybranie algorytmu, za pomocą którego testowana

jest dana wejściowa. Dodatkowo struktura zawierająca metody pozwalające na operacje matematyczne dla liczb o dowolnej długości w języku C++ została napisana samodzielnie.

1.2 Podstawy teoretyczne

1.2.1 Test pierwszości Fermata

Test pierwszości Fermata, to probabilistyczny test pierwszości. Oznacza to, że określa on czy wprowadzona liczba jest prawdopodobnie pierwsza czy złożona. U podstaw tego algorytmu leży tzw. małe twierdzenia Fermata, które zostało przedstawione poniżej.
Małe twierdzenie Fermata
Niech $a \in \mathbb{Z}$, wtedy

Jeśli liczba p jest liczbą pierwszą to dla dowolnego a, liczba $a^{p-1} - a$ jest podzielna przez p.

$$a^p - a \equiv 0 \,(mod\, p)$$

Sam algorytm opiera się o wykonanie zadaną ilość razy poniższych czynności:

1. Wylosuj liczbę a, takie że $1 < a < p$.
2. Sprawdź czy $a^{p-1} \,mod\, p = 1$.
3. Jeżeli nie to przerwij test - liczba nie jest pierwsza.

Jeśli równość była spełniona dla zadanej liczby prób - liczba jest prawdopodobnie pierwsza.

W tym przypadku największym problemem staje się obliczenie wyrażenia $a^{p-1} \,mod\, p$. Rozwiązaniem tego problemu jest wykorzystanie algorytmu szybkiego potęgowania modularnego którego poszczególne etapy zostały przedstawione poniżej.

Szybkie potęgowanie modularne

Niech dane będzie wyrażenie postaci $a^b \, mod \, n$, wtedy

1. Rozłóż liczbę b na jej postać binarną $b = (b_m, b_{m-1}, ..., b_1, b_0)$, gdzie b_0 oznacza najmłodszy bit.

2. Wtedy początkowe wyrażenie przedstawić możemy w następujący sposób
   $a^b \, mod \, n = (a^{b_0 * 2^0} \, mod \, n) * (a^{b_1 * 2^1} \, mod \, n) * ... * (a^{b_m * 2^m} \, mod \, n)$

3. Należy skorzystać z własności, że $a^{2^m} \, mod \, n = (a^{2^{m-1}} * a^{2^{m-1}}) \, mod \, n$, a także z obserwacji, że jeżeli bit x jest równy 0, to $(a^{b_x * 2^x} \, mod \, n) = (a^{0 * 2^x} \, mod \, n) = (a^0 \, mod \, n) = 1$.

4. Przemnóż przez siebie tylko te potęgi, którym odpowiada bit 1 w liczbie b.

1.2.2 Test pierwszości Millera-Rabina

Test pierwszości Millera-Rabina, podobnie jak wyżej opisany test pierwszości Fermata, jest algorytmem probabilistycznym. Opiera się on o następujące twierdzenie.

Twierdzenie 1

Niech dana będzie nieparzysta liczba pierwsza p postaci $p = 1 + 2^s d$, gdzie d jest liczbą nieparzystą. Wtedy dla dowolnej liczby naturalnej $a \in [2, p-2]$, ciąg Millera-Rabina postaci $a^d, a^{2d}, a^{4d}, ..., a^{2^{s-1}d}, a^{2^s d} (mod \, p)$ kończy się liczbą 1.
Dodatkowo, jeśli a^d nie przystaje modulo p do 1, to wyraz ciągu Millera-Rabina bezpośrednio poprzedzający 1 jest równy $p - 1$.

Test Millera-Rabina daje złe wyniki dla co najwyżej $\frac{1}{4}$ baz $a < p$. W związku z tym prawdopodobieństwo błędu dla jednego przebiegu algorytmu wynosi $\frac{1}{4}$. Jeżeli dla danej wejściowej powtórzymy test k-krotnie, przy różnych podstawach a to prawdopodobieństwo błędu spada do $(\frac{1}{4})^k$. W związku z tym możemy sprawdzić z bardzo dużym prawdopodobieństem czy liczba p jest liczbą pierwszą.

Sam algorytm przedstawić można w następujący sposób.
Niech k - określa dokładność testu Millera-Rabina, a n - będzie testowaną liczbą nieparzystą, wtedy:

1. Wylicz s - maksymalną potęgę dwójki dzielącą $n-1$
2. Podstaw $d = \frac{n}{2^s}$.
3. Powtarzaj k razy:
4. Wylosuj a takie, że $1 < a < n$
5. Sprawdź czy $a^d \, mod \, n \neq 1$
6. Jeśli tak, to sprawdź czy $a^{d*2^r} \, mod \, n \neq n-1$ dla wszystkich r, takich że: $0 \leq r \leq s-1$
7. Jeśli tak, to przerwij test - liczba nie jest pierwsza
8. Jeśli nie przerwano testu dla żadnej z prób to liczba prawdopodobnie jest pierwsza

Podobnie jak w przypadku testu pierwszości Fermata, problemem może okazać się obliczenie wyrażenia $a^d \, mod \, n$, jednakże w celu przyspieszenia działania algorytmu należy analogicznie jak we wcześniej opisanym teście pierwszości skorzystać z algorytmu szybkiego potęgowania modularnego.

1.2.3 Test pierwszości Agrawal-Kayal-Saxena

Jest to deterministyczny test pierwszości opublikowany przez trzech badaczy z Indian Institute of Technology Kanpur w 2002r. Algorytm ten określa czy zadana liczba jest pierwsza, czy złożona w czasie wielomianowym. Test pierwszości AKS korzysta z jednego twierdzenia oraz faktu, które zostały opisane poniżej.

Twierdzenie 1

Niech $n \geqslant 2$ i n jest liczbą całkowitą, oraz a jest również liczbą całkowitą względnie pierwszą z n, wtedy:

$$(X - a)^n \equiv (X^n - a) \, (mod \, n),$$

gdzie X - formalny symbol, oznaczający przestrzeń wielomianów.

Fakt 1

Niech n będzie liczbą pierwszą, wtedy dla wszystkich $0 < k < n$, prawdziwa jest zależność:

$$\binom{n}{k} \equiv 0 \, (mod \, n)$$

Istnieje wiele rozwinięć algorytmów AKS, a każda z nich modyfikuje część etapów podstawowego algorytmu w celu przyspieszenia jego działania. Zaimplementowana w programie wersja algorytmu to algorytm AKS z udogodnieniami wprowadzonymi przez Hendrika Lenstra. Sam algorytm w bardo uogólniony sposób przedstawić można w pięciu krokach, opierając się o założenia przedstawione w sekcjach "Twierdzenie 1", oraz "Fakt 1".

1. Należy sprawdzić czy $n = a^b$ dla $a \in \mathbb{N}$ i $b > 1$. Jeżeli tak to wprowadzona liczba n nie jest pierwsza.

2. W tym kroku należy znaleźć liczbę r, która posłuży w dalszej części algorytmu AKS. Najmniejszą liczbę r szuka się ze wzoru $o_r(n) > (log_2(n))^2$. Gdzie $o_r(n)$ jest równe najmniejszemu k, które to k wyznaczyć można ze wzoru $n^k = 1(mod \, r)$. Nierówność ta wykonywana jest do czasu znalezienia takiego r, które spełnia tą nierówność.

3. Krok trzeci polega na wyliczeniu podwójnej nierówności postaci $1 < (a, n) < n$. W tym kroku następuje sprawdzenie czy kolejne liczby naturalne a, spełniają warunek

względnej pierwszości z n. Jeśli tak to spełnione są oba warunki nierówności. Należy również sprawdzić czy $a \leqslant r$. Jeżeli nie istnieje takie a, można przejść do następnego kroku.

4. Krok ten polega na sprawdzeniu, czy liczba $n \leqslant r$. Jeśli warunek jest spełniony, należy przejść do następnego kroku.

5. Ostatni krok polega na sprawdzeniu liczb a z przedziału $(1, |\sqrt{\phi(r)} log_2(n)|)$, gdzie $\phi(r)$ to funckja Eulera określająca ilość liczb względnie pierwszych z r, i podstawieniu ich do podanej równości.

$$(X + a)^n = X^n + a (mod\, X^r - 1\,, n)$$

Przy czym wartości w nawiasie oznaczają dzielenie modulo najpierw przez n, a następnie przez wielomian $X^r - 1$. Natomiast wielomian $(X + a)^n$, obliczany jest na podstawie poniższego wzoru.

$$(X + a)^n = \binom{n}{0} X^n + \binom{n}{1} X^{n-1} a + \binom{n}{2} X^{n-2} a^2 + ... + \binom{n}{n} a^n$$

Jeżeli po przejściu całego zakresu liczby a, każde z równań jest spełnione, wtedy zwróć informację, że liczba jest pierwsza.

2 Opis rozwiązania

Ze względu na dużą ilość przyjętych rozwiązań w końcowym programie - sekcję tą pragnę przedstawić w następujący sposób. Rozdział ten rozłożony jest na opis każdego z zaimplementowanych algorytmów. W każdym z nich opisywać będę jedynie najistotniejsze elementy, wraz z przytoczonymi fragmentami kodu źródłowego. W przypadku elementów dotyczących wszystkich lub większości wymienionych algorytmów - zostały one opisane poniżej. W momencie wystąpienia powtórzenia któregoś z rozwiązań (np. szybkie potęgowanie modularne wykorzystywane w teście pierwszości fermata i millera-rabina), będę jedynie odnosił się do niego, bez ponownego opisywania.

Głównym trzonem całego programu jest struktura “BigInt”, która została przeze mnie zaimplementowana. W skład tej struktury wchodzą wszystkie niezbędne funkcje oraz zmienne potrzebne do późniejszych obliczeń. Głównym założeniem jest to by wprowadzona liczba reprezentowana była w postaci wektora liczb typu "int", a jedna liczba w wektorze składała się z maksymalnie dziewięciu cyfr. Oznacza to, że wprowadzona liczba, np. "1234567891111111111", jest reprezentowana w postaci
$wektor[0] =$ ”123456789”, $wektor[1] =$ ”111111111”. Rozwiązanie to ma na celu przyspieszenie wykonywania obliczeń.

2.1 Test pierwszości Fermata

Test pierwszości fermata w zaimplementowanej wersji opiera się głównie o algorytm szybkiego potęgowania modularnego. Poniżej przedstawiam funckję odpowiedzialną za przeprowadzenie testu pierwszości Fermata, oraz funkcję obliczającą szybkie potęgowanie modularne.

Listing 1: Funkcja wykonująca test pierwszości Fermata

```
1  bool checkFermatPrime(int repeats, vector<int> num1) {
2          vector<int> num1Decrement = substract(num1, ONE);
3          for (int i = 0; i < repeats; i++) {
4              int random = generateRandom(num1);
5
6              if (compareEqual(powerFastModulo(Integer(to_string(random)),
                   num1Decrement, num1), ONE))
7                  continue;
8              else
9                  return false;
10         }
11         return true;
12     }
```

Funkcja ta przyjmuje ilość powtórzen testu, oraz liczbę w postaci wektora liczb. W liniach nr 2 i 4 przygotowywane są zmienne lokalne potrzebne do wykonania testu. Funkcja "generateRandom(num1)", to samodzielnie napisana funkcja, która jako argument przyjmuje liczbę w postaci wektora i zwraca losową liczbę z przedziału $[2, num1 - 1]$. Samo sprawdzenie przy pomocy algorytmu szybkiego potęgowania modularnego następuje w linii nr 6. Funkcja "compareEqual" porównuje czy wartość po lewej stronie - "powerFastModulo(Integer(to_string(random)),num1Decrement,num1)", równa jest 1. Funkcja "powerFastModulo" została opisana poniżej.

Listing 2: Funkcja wykonująca szybkie potęgowanie modularne

```
1  vector<int> powerFastModulo(vector<int>a, vector<int>b, vector<int>n) {
2          vector<int> x = a;
3          vector<int> result = ONE;
4          vector<int> bCopy = b;
5          bool run = true;
6          int counter = 0;
7          do {
8              b = bCopy;
9              if (b[b.size() - 1] % 2 == 1) {
10                 result = multiply(result, x);
11                 result = modulo(result, n);
12             }
13             x = multiply(x, x);
14             x = modulo(x, n);
15             bCopy = divide(b, TWO);
16             counter++;
17         } while (!compareGreater(ONE, bCopy));
18         return result;}
```

Funkcja ta przyjmuje argumenty w postaci wektorów liczb odpowiadających liczbom we wzorze $a^b \, mod \, n$. W liniach od 2 do 7 przygotowywane są zmienne lokalne potrzebne do obliczeń oraz kontroli przebiegu funkcji. Docelowe obliczenia wykonywane są w pętli w liniach od 7 do 17. Najciekawsza z operacji ma miejsce w linii 9, ponieważ to właśnie tu liczba b rozkładana jest na postać binarną poprzez operację dzielenia modulo 2. Jeżeli na danej pozycji występuje biarne 1, wtedy warunek zostaje spełniony, a końcowy wynik zostaje pomnożony przez aktualną potęge a (linia 10) i podzielona modulo przez wartość n. W przeciwnym wypadku zostają wykonane operacje na kopii liczby a, by były one gotowe do użycia przy następnym przebiegu pętli.

Listing 3: Funkcja wykonująca test pierwszości Fermata w języku Python(3.8)

```
1  def checkFermatPrime(args):
2      i=0
3      repeats=20
4      number = int(args)
5      while i<repeats:
6          a = random.randint(1,number-1)
7          if pow(a,(number-1),number) != 1:
8              return False
9          i = i+1
10     return True
```

Przedstawiona powyżej funkcja jest implementacją tego samego algorytmu wykorzystanego w przypadku listingu nr 1, jednakże wykonana została w języku Python. Jako argument przyjmuje ona zadaną liczbę do przetestowania jako "args". Istnieją zasadnicze dwie różnice pomiędzy implementacją w języku C++, a Python. Pierwszą z nich jest brak przyjmowanej liczby powtórzeń testu dla badanej liczby, natomiast drugim bardziej istotnym faktem jest wykorzystanie systemowej funkcji "pow", języka Python. Funkcja "pow" wywołana z trzema argumentami wykonuje operację szybkiego potęgowania modularnego, przez co nie jest konieczne jej ręczne implementowanie.

2.2 Test pierwszości Millera-Rabina w wersji probabilistycznej

Test pierwszości Millera-Rabina w wersji probabilistycznej opiera się w głównej mierze o ten sam algorytm szybkiego potęgowania modularnego, który to został przedstawiony w sekcji poświęconej testowi pierwszości Fermata. Jednakże, zawiera on też funkcję odpowiedzialne za obliczenie maksymalnej potęgi dwójki dzielącej liczbę $n-1$, gdzie n jest testowaną liczbą, a także obliczającej paramter d, który jest ilorazem wprowadzonej liczby na pozycji dzielnej i liczby 2^s, na pozycji dzielnika. Parametr ten wykorzystywany jest w dalszej części algorytmu. Poniżej przedstawione zostały listingi wyżej opisanych funkcji.

Listing 4: Funkcja wykonująca test pierwszości Millera-Rabina

```
1   bool checkPrime(int repeats, int s, vector<int> d, vector<int> mod) {
2       vector<int> one = ONE;
3       vector<int>modDecrement = substract(mod, one);
4       for (int i = 0; i < repeats; i++) {
5           int a = generateRandom(mod);
6           string A = to_string(a);
7           if (!compareEqual(powerFastModulo(Integer(A), d, mod), one)) {
8               for (int r = 0; r <= s; r++) {
9                   vector<int>temporary = multiply(Integer(to_string(int(
                        pow(2, r)))), d);
10                  if (!compareEqual(powerFastModulo(Integer(A), temporary,
                         mod), modDecrement)) {
11                      if (r == (s - 1))
12                          return false;
13                  }
14                  else
15                      break;
16              }
17          }
18      }
19      return true;}
```

Funkcja ta przyjmuje liczbę powtórzeń wykoania testu, wynik funkcji obliczonej wartości "s", wartość obliczonego parametru "d", oraz liczbę "mod", która poddana będzie testowi. Funkcja ta bazuje na krokach wypisanych w rozdziale 1.2.2. W celu obliczenia wartości "s" i "d", należy posłużyć się funkcjami, których listingi zostały przedstawione poniżej.

Listing 5: Funkcja obliczająca maksymalną potęgę dwójki dzielącą liczbę $number - 1$

```
1   int findS(vector<int>number) {
2     int maxPow = 0;
3     vector<int> numberDecrement = substract(number, ONE);
4     while (true) {
5         string Power = "";
6         int power = 0;
7         power = int(pow(2, maxPow));
8         Power = to_string(power);
9         bool equal = compareEqual(modulo(numberDecrement, Integer(Power
              )), ZERO);
10        if (equal) {
11            maxPow++;
12            continue;
13        }
14        else {
15            break;
16        }
17    }
18    maxPow--;
19    return maxPow;}
```

Metoda obliczenia parametru "s", oparta jest na dzieleniu modulo podanej liczby przez kolejne potęgi dwójki. Pętla ta wykonywać ma się tak długo dopóki dzielenie modulo będzie równe 0. W momencie, gdy wynik będzie różny od 0, następuje wyjście z pętli i zwrócenie wartości o jeden mniejszej.

Listing 6: Funkcja obliczająca całkowitoliczbowy iloraz postaci $\frac{num1}{2^s}$

```
1       vector<int> findD(vector<int> num1, int s) {
2        vector<int> result = divide(num1, power(TWO, Integer(to_string(s)))
            );
3        return result;}
```

Funkcja ta przyjmuje zadaną liczbę "num1", oraz parametr "s", obliczony przy pomocy funkcji "findS". Następnie w drugiej linii następuje dzielenie podanych wartości, a dalej zwracany zostaje iloraz, czyli parametr "d", używany przez algorytm Millera-Rabina.

2.3 Test pierwszości Millera-Rabina w wersji deterministycznej

Jedyne różnice występujące pomiędzy deterministyczną, a probabilstyczną wersją algorytmu Millera-Rabina to:

- Algorytm w wersji deterministycznej działa na ustalonych wartościach liczby a (rozdział 1.2.2), jedynie dla $n < 18\,446\,744\,073\,709\,551\,616$. W przeciwnym razie wykonywana jest wersja probabilistyczna algorytmu.
- Z uwagi na z góry ustalone wartości liczby a, algorytm wykonywany jest również z góry ustaloną liczbę razy.

W związku z powyższym, poniżej przedstawiam fragment funkcji w języku Python pórwnującej wprowadzoną liczbę i przydzielającą jej odpowiednie wartości liczby a. Analogiczna funkcja została napisana dla wersji algorytmu w języku C++.

Listing 7: Funkcja przydzielająca odpowiednie wartości liczbie a w algorytmie MR

```
1 def checkSection(number):
2     sections = [0,2047,1373653,9080191,25326001,3215031751, 4759123141,
3     1122004669633,2152302898747,3474749660383,341550071728321,
4     3825123056546413051,18446744073709551616]
5     if number == sections[0]:
```

```
6         return []
7     elif(number > sections[0] and number < sections[1]):
8         return [2]
9     elif(number >= sections[1] and number < sections[2]):
10        return [2,3]
11    elif(number >= sections[2] and number < sections[3]):
12        return [31,73]
13    ...
14    elif(number >= sections[11] and number < sections[12]):
15        return [2,3,5,7,11,13,17,19,23,29,31,37]
16    else:
17        returnValues=[]
18        for i in range(0,20,1):
19            returnValues.append(random.randint(1,number-1))
20        return returnValues
```

Funkcja jako argument przyjmuje podaną liczbę, a następnie sprawdza przedział liczbowy do którego należy podana liczba. Odpowiednie wartości graniczne dla poszczególnych przedziałów znajdują się w strukturze "sections". Jeżeli podana liczba mieści się w zakresie poszczególnego przedziału - zwracana jest lista liczb, które zostaną w odpowiedniej kolejności przypisane do wartości a (losowej liczby) w algorytmie Millera-Rabina. Jeżeli jednak dana liczba jest większa niż największa przewidziana wartość - losowanych jest dwadzieścia liczb z zakresu od 1 do $number - 1$, umieszczane są w odpowiedniej tablicy (linia 19), a następnie są one zwracane.

2.4 Test pierwszości Agrawal-Kayal-Saxena

Test ten oparty jest o pięć kroków, które zostały opisane w rozdziale 1.2.3. Poniżej przedstawię poszczególne etapy, wykonane w języku Python.

Listing 8: Funkcja wykonująca pierwszy krok algorytmu AKS

```
1 def firstStep(n):
2     for b in range(2,int(math.floor((math.log2(n)+1))), 1):
3             a = n**(1/b)
4             if math.floor(a) == a:
5                     return False
6     return True
```

Funkcja ta przyjmuje wprowadzoną przez użytkownika liczbę, a nastepnie na podstawie przekształconego równania $n = a^b$, sprawdza czy dana liczba nie jest złożona. Jeśli iterator pętli sprawdzi cały dostępny zakres b, zwracany jest obiekt True oznaczający, że podana liczba nie jest złożona. Przekształcenie funkcji jakie zostało wykorzystane w wyżej opisanej funkcji przedstawiam poniżej.

$$n = a^b \,/\, log_2$$
$$log_2(n) = b * log_2(a) / : b$$
$$\frac{log_2(n)}{b} = log_2(a) \Leftrightarrow 2^{\frac{log_2(n)}{b}} = a, \text{ gdzie } 2^{\frac{log_2(n)}{b}} = n^{\frac{1}{b}}, \text{ otrzymując}$$
$$n^{\frac{1}{b}} = a$$

Następny krok dotyczył znalezienia takiej liczby r, która spełnia nierówność $o_r(n) > (log_2(n))^2$. Funkcję odpowiedzialną za znalezienie takiej wartości r prezentuję poniżej.

Listing 9: Funkcja wykonująca drugi krok algorytmu AKS

```
1 def secondStep(n):
2         minK = math.floor(math.log2(n)**2)
3         nextr = True
4         r = 1
5         while nextr is True:
6             r += 1
7             nextr = False
```

```
8           k = 0
9           while k <= minK and nextr is False:
10              k = k+1
11              if pow(n, k, r) in (0, 1):
12                  nextr = True
13      return r
```

Funkcja ta działa na zasadzie opisanej w kroku drugim w rozdziale 1.2.3, tj. obliczane jest najmniejsze k, spełniające równanie $n^k = 1(mod\, r)$, a następnie sprawdzane jest czy otrzymane k jest większe od $(log_2(n))^2$.

Następny krok opiera się o sprawdzenie, czy kolejne liczby naturalne a, spełniają warunek względnej pierwszości z n. W tym celu posłużyłem się dostępną w module "math", funkcją gcd.

Listing 10: Funkcja wykonująca trzeci krok algorytmu AKS

```
1 def thirdStep(n, r):
2     for a in range(1, r+1, 1):
3             if((1 < math.gcd(a,n)) and (math.gcd(a,n) <n)):
4                     return False
5     return True
```

Następnym krokiem było proste sprawdzenie, czy liczba n jest mniejsza lub równa r. Kod tej funkcji prezentuję poniżej.

Listing 11: Funkcja wykonująca czwarty krok algorytmu AKS

```
1 def fourthStep(n, r):
2     if(n <= r):
3             return True
4     else:
5             return False
```

Jeżeli warunek jest spełniony, podana liczba jest liczbą pierwszą i nie trzeba przechodzić do ostatniego kroku algorytmu AKS. W przeciwnym razie wywoływana jest piąta funkcja wchodząca w skład algorytmu AKS, której kod prezentuję poniżej.

Listing 12: Funkcja wykonująca piąty krok algorytmu AKS

```
1  def fifthStep(n, r):
2      phiN = eulerFunction(r)
3      maxRange = math.floor(math.sqrt(phiN)*math.log2(n))
4      pythonManager = multiprocessing.Manager()
5      sharedDictionary = pythonManager.dict()
6      processes = []
7      for a in range(0, maxRange, 1):
8          proces = multiprocessing.Process(target=handlePolynomials, args=(n,
               a, a+1, sharedDictionary))
9          proces.start()
10         processes.append(proces)
11     for Pn in processes:
12         Pn.join()
13     if False not in sharedDictionary.values():
14             return True
15     else:
16             return False
```

Funkcja ta pośrednio korzysta z dwóch dodatkowych funkcji. Pierwszą z nich jest funkcja "eulerFunction(r)", która służy do wyznaczenia liczbie r liczbę liczb względnie pierwszych z nią i nie większych od niej. Jest ona potrzebna do obliczenia $|\sqrt{\phi(r)} * log_2(n)|$, które to wyrażenie obliczane jest w trzeciej linii funkcji.Linie 4,5,6 odpowiedzialne są za przygotowanie odpowiednich obiektów do działań z zastosowaniem mechanizmu multiprocessingu. Od 7 linii następuje w kolejności - stworzenie osobnego procesu, który uruchomić ma funkcję "handlePolynomials" z odpowiednimi argumentami w celu obliczenia poszczególnych

wielomianów cząstkowych na podstawie wzoru $(X + a)^n$. Następnie proces ten jest uruchamiany i dodawany do listy procesów. W liniach 11 i 12 pętla iteruje po wszystkich obiektach znajdujących się na liście procesów i "nakazuje" im poczekać dopóki nie skończą obliczeń. Następnie w liniach 13-20, sprawdzany jest współdzielony słownik, w celu natrafienia na wartość False, oznaczającą niepomyślne wykonanie obliczeń na wielomianach. Jeżeli wartość ta nie wystąpi, zwracany jest obiekt typu True, oznaczający, że dana liczba jest pierwsza. W przeciwnym razie zwracany jest obiekt typu False. Poniżej prezentuję w kolejności funkcję odpowiedzialną za obliczenie funkcji Eulera oraz funckję operującą na wielomianach cząstkowych.

Listing 13: Funkcja obliczająca funkcję Eulera dla danego n

```
1  def eulerFunction(n):
2      result = 0
3      for i in range(1, n + 1, 1):
4              if(math.gcd(i, n) == 1):
5                      result += 1
6      return result
```

Listing 14: Funkcja tworząca i wykonująca operacje modulo na zadanych wielomianach

```
1  def handlePolynomials(number, n, k, sharedDictionary):
2      power = n/(k-n)
3      if n == 0:
4          n = 1
5      for a in range(n,k,1):
6          b = power(a,n,n)
7          if ((b-a) != 0):
8              sharedDictionary[x] = False
9              return False
10     sharedDictionary[x] = True
```

```
11      return True
```

Ostatnią funkcją, jaką pragnę przedstawić jest funkcja obliczająca całość algorymu AKS, tj. składająca się z poszczególnych wyżej opisanych etapów.

Listing 15: Funkcja wykonująca algorytm AKS

```
1  def Aks(args):
2          args = int(args)
3          if(firstStep(args) is True):
4                  r = secondStep(args)
5                  if (thirdStep(args,r) is True):
6                          if (fourthStep(args, r) is True):
7                                  print('Liczba jest pierwsza')
8                          else:
9                                  if(fifthStep(args,r) is True):
10                                     print('Liczba jest pierwsza')
11                                 else:
12                                     print('Liczba nie jest pierwsza')
13                 else:
14                         print('Liczba nie jest pierwsza')
15         else:
16                 print("Liczba nie jest pierwsza")
```

3 Analiza porównawcza

W tej sekcji pragnę przedstawić wyniki analizy porównawczej następujących testów pierwszości:

- Millera-Rabina w wersji probabilistycznej,
- Millera-Rabina w wersji deterministycznej,
- Fermata

Wyniki wykonywanych algorytmów rozpatrywane będą pod względem czasowym oraz jakościowym w skład którego wchodzi porównanie ilość powtórzeń testu potrzebnych na stwierdzenie czy liczba jest prawdopodobnie pierwsza oraz podatność algorytmów na tzw. liczby Carmichaela. Wszystkie pomiary wykonywane były na komputerze o poniższej specyfikacji:

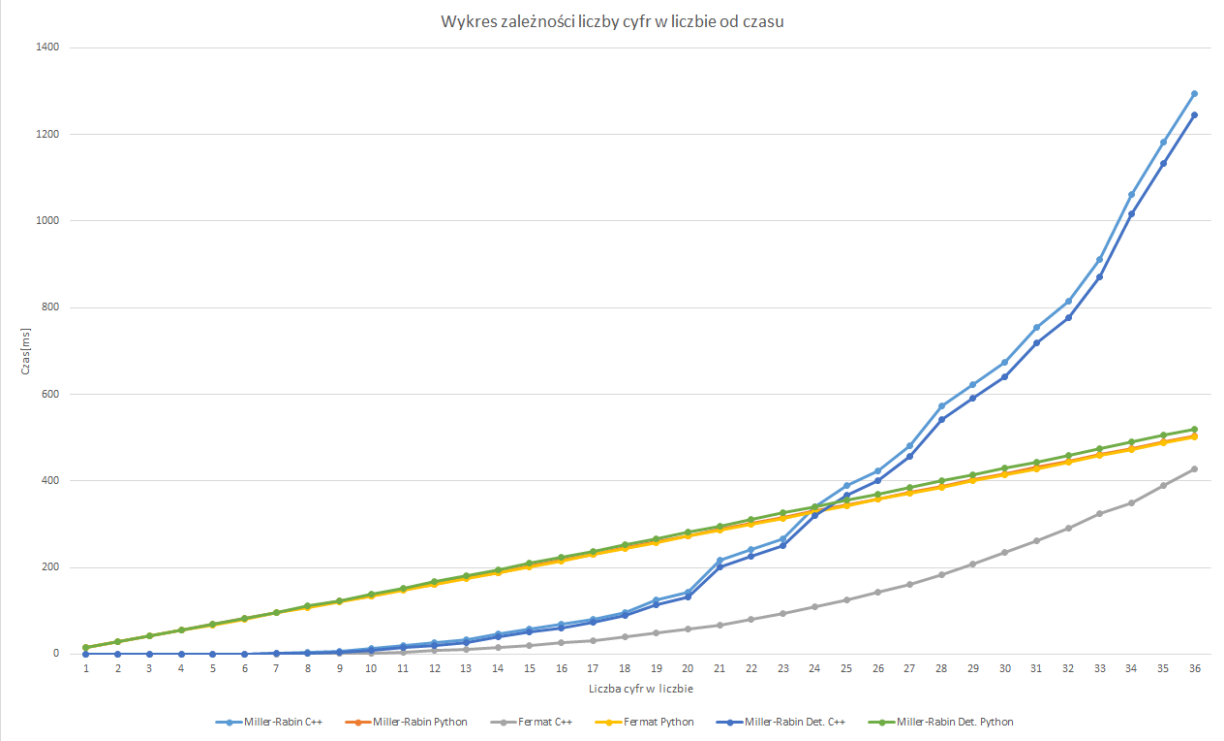
- Procesor sześciordzeniowy - Intel Core i5 8600K - o częstotliwości taktowania 3.60GHz.
- Pamięć RAM typu DDR4 o łącznej pamięci 8GB i częstotliwości 2666MHz.

Każdy z algorytmów uruchamiany był dwudziestokrotnie, niezależnie od ilości przebiegów pętli wewnętrznych, a na ich podstawie zostały wykonane statystyki pomiarów. Dodatkowo dla każdego z algorytmu wprowadzane były stałe rodzaje danych w postaci tablic liczb zawierających odpowiednio: liczby nieparzyste w których żadna z nich nie była pierwsza, liczby nieparzyste w których wszystkie były liczbami pierwszymi, oraz ostatnia tablica zawierająca liczby pierwsze, liczby nie będące pierwszymi, a także liczby Carmichaela.

W ostatniej z wyżej wymienionych tablic liczby Carmichaela składały się z odpowiednio: 3,4,5,16 oraz 19 cyfr. Liczby pierwsze składały się natomiast z 1,2,6,9,14 oraz 22 cyfr. Pozostała zawartość tablicy wypełniona była liczbami nieparzystymi nie będącymi liczbami pierwszymi. Dodatkowo pętle wewnętrzne algorytmów (poza deterministyczną wersją algorytmu Millera-Rabina) testowane były dla odpowiednio 5, 10 i 20 powtórzeń.

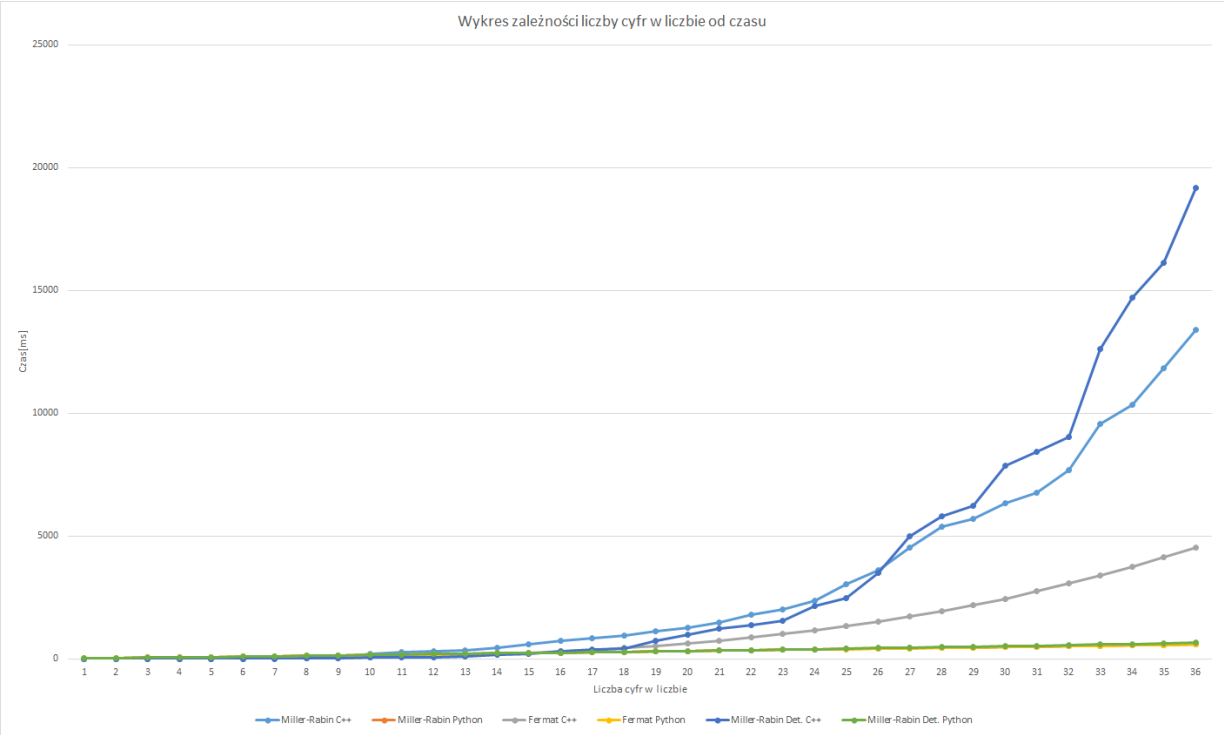
3.1 Analiza Czasowa

Pierwsze porównanie algorytmów jakie pragnę przedstawić dotyczy ich prędkości działania. Ze względu na podobieństwo kształtu krzywych na wykresach posłużę się jedynie przypadkiem dla dziesięciu powtórzeń pętli wewnętrznych.



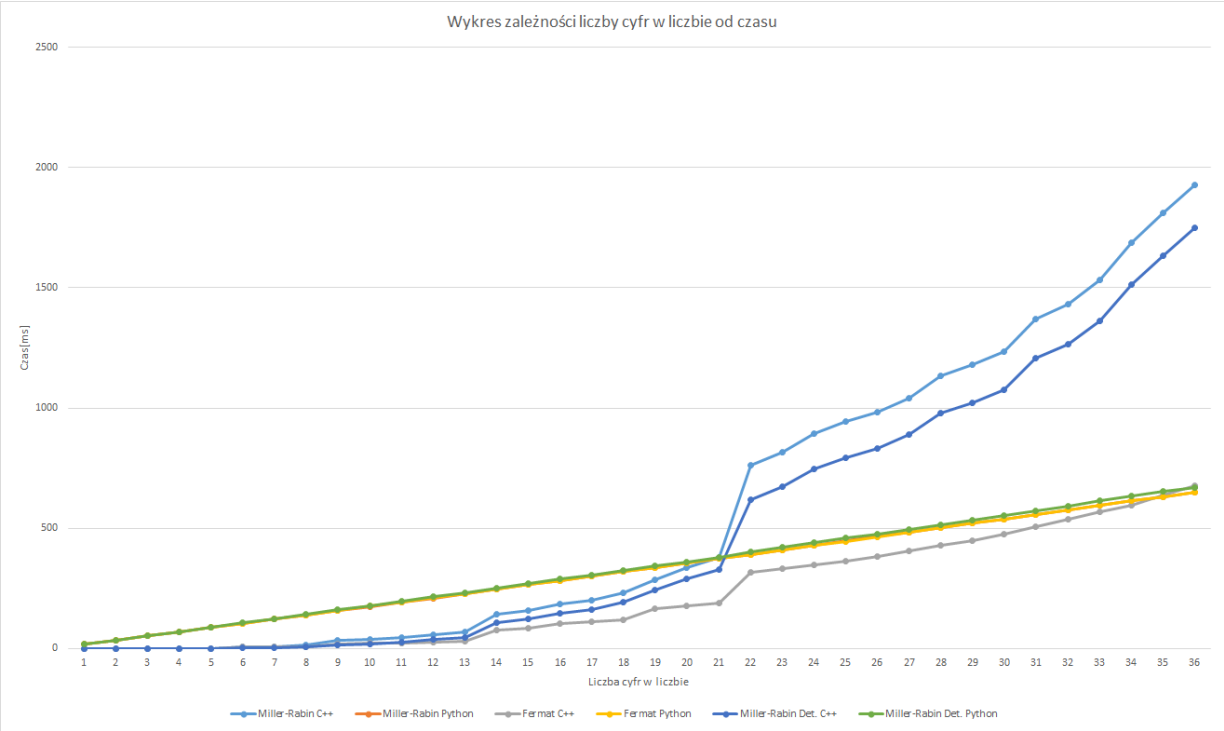
Rysunek 1: Wykres zależności liczby cyfr w liczbie od czasu dla liczb nie będących pierwszymi

Na podstawie powyższego wykresu można zobaczyć, że algorytmy napisae w języku C++ pod względem czasowym mają znaczącą przewagę w prędkości działania dopóki liczba będzie zawierała mniej niż 23 cyfry. Zaobserwować możemy dodatkowo tendecję w jaką układają się krzywe na wykresie. Algorytmy napisane przy użyciu języka Python od samego początku wykazują kształt charakterystyczny dla funkcji liniowych. Z drugiej strony mniej spodziewany są kształty krzywych algorytmów napisanych przy użycia języka C++, które zbliżone są do funkcji wykładniczych.



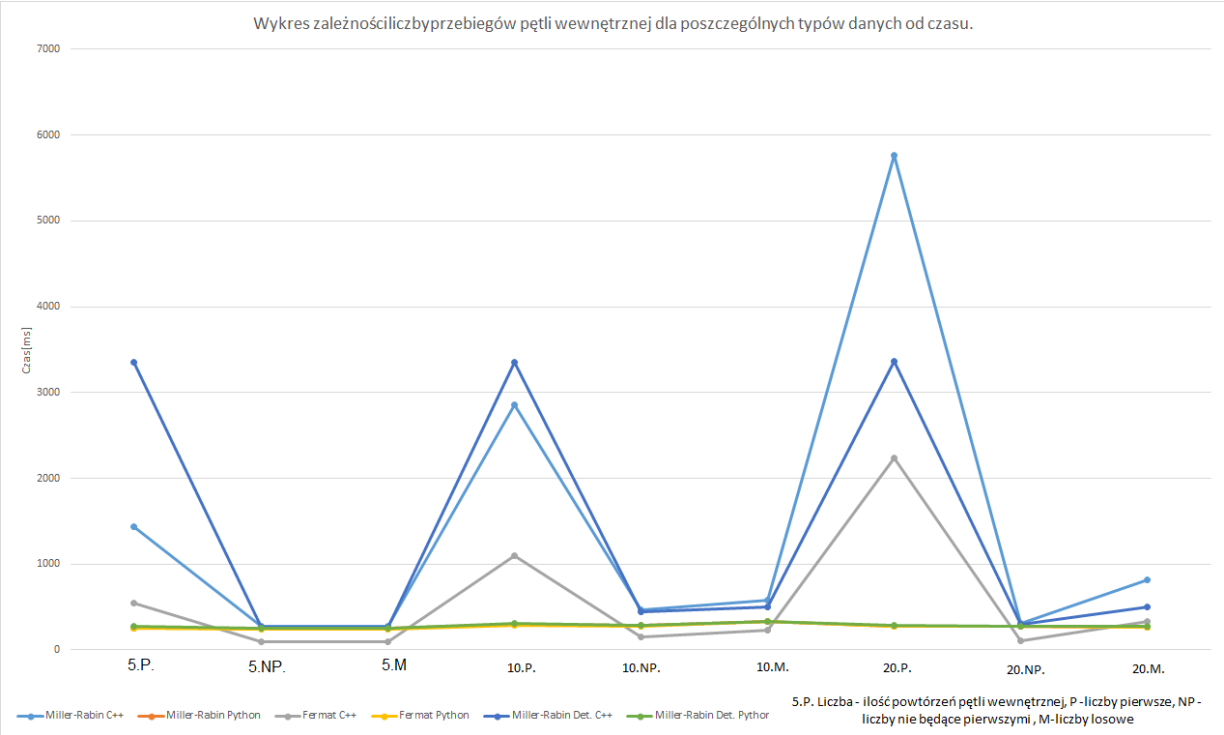
Rysunek 2: Wykres zależności liczby cyfr w liczbie od czasu dla liczb pierwszych

Również w przypadku testowania algorytmów dla liczb pierwszych algorytmy napisane w języku Python charakteryzują się liniową tendecją wzrostową w odróżnieniu od funkcji napisanych przy pomocy języka C++, dla których krzywa wciąż charakteryzuje się wykładniczą tendencją wzrostową. Jedynie dla liczb składających się z co najwyżej ok. 10 cyfr, można przyjąć, że wszystkie z badanych algorytmów wykazują podobny czas działania.



Rysunek 3: Wykres zależności liczby cyfr w liczbie od czasu dla losowych liczb

Przedstawiony powyżej wykres jest jednym z ciekawszych, ponieważ został on przeprowadzony na pseudolosowych liczbach. Wiedząc dokładnie, które liczby były pierwsze, a które nimi nie były (opisane na stronie 21), zaobserwować możemy wzrost czasu działania algorytmu w tych konkretnych przypadkach. Ciekawym spostrzeżeniem wydaje się być fakt, że algorytmy wydają się być niewrażliwe na liczby Carmichaela oraz to, że do liczb składających się z co najwyżej 19 cyfr szybszym działaniem okazują się być algorytmy napisane w języku C++, tak samo jak ma to miejsce w przypadku testowania algorytmów na samych liczbach nie będących liczbami pierwszymi.



Rysunek 4: Wykres przedstawiający różnice w prędkości działaniu algorytmów dla różnej liczby powtórzeń pętli wewnętrznych algorytmów.

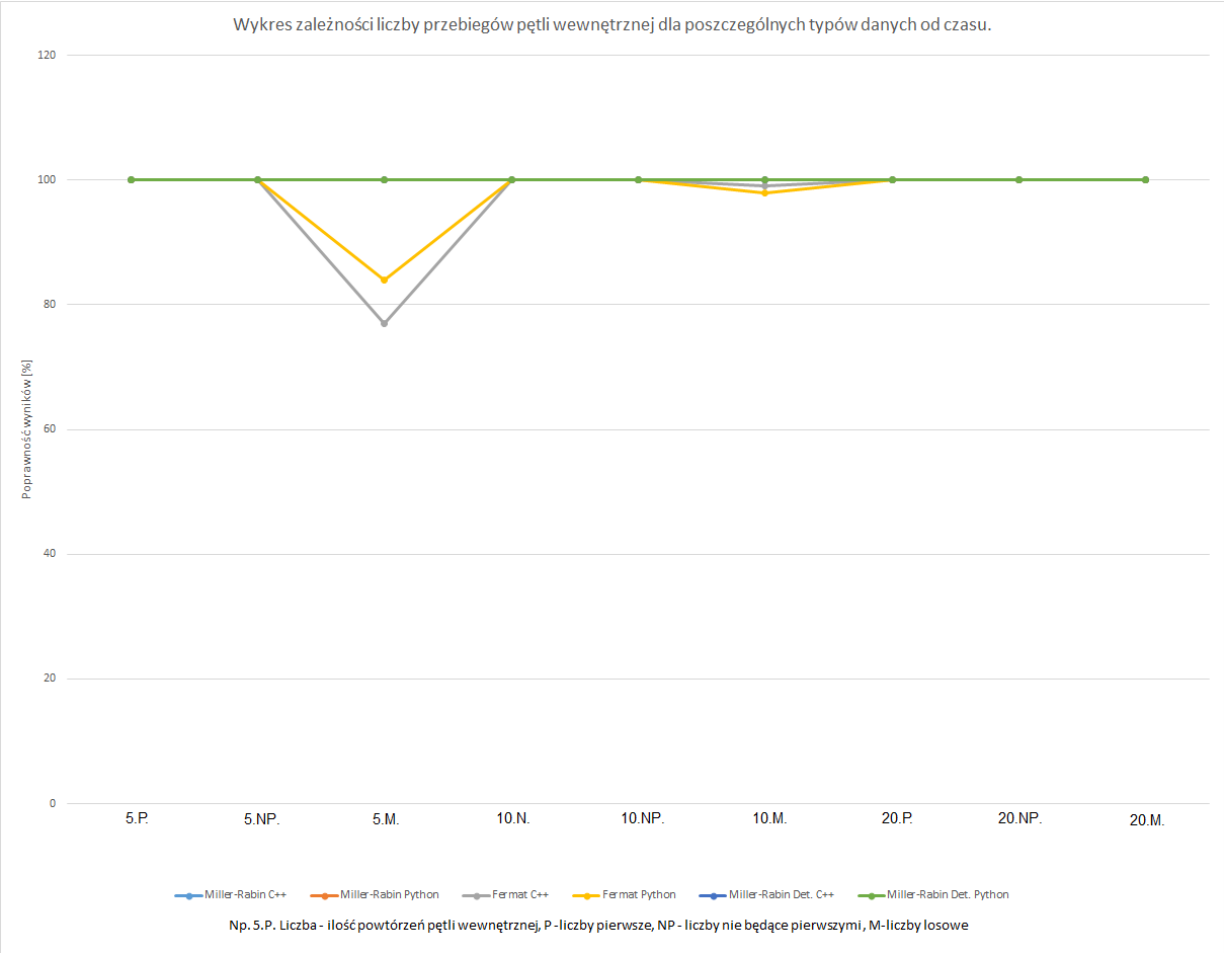
Na wyżej przedstawionym wykresie zaobserwować można porównanie algorytmów, pod względem zmiennego ustawienia ich pętli wewnętrznych. Najciekawszymi spostrzeżeniami jakie można zobaczyć to m.in. liniowy czas działania algorytmu Millera-Rabina w wersji deterministycznej napisanego w języku C++ dla liczb pierwszych, oraz w przybliżeniu liniowy czas działania dla wszystkich algorytmów napisanych w języku Python, niezależnie od danych wejściowych. Dodatkowo warto zwrócić uwagę na proporcjonalny wzrost czasu dla algorytmu Millera-Rabina w języku C++ w momencie podania liczby pierwszej na wejście funkcji.

Na podstawie analizy wszystkich powyższych wykresów wyciągnąć można następujące wnioski:

- Algorytmy napisane w języku C++, wykonują obliczenia w krótszym czasie w porównaniu do algorytmów napisanych w języku Python, jedynie dla liczb składających się z co najwyżej 19 cyfr.
- Algorytmy napisane w języku C++ charakteryzują się wykładniczą tendencją wzrostową, natomiast algorytmy napisane przy pomocy języka Python charakteryzują się liniową tendencją wzrostową.
- Algorytmy napisane w języku Python wydają się być niewrażliwe pod względem czasu wykonywania działania na dane wejściowe w przeciwieństwie do algorytmów napisanych w C++.

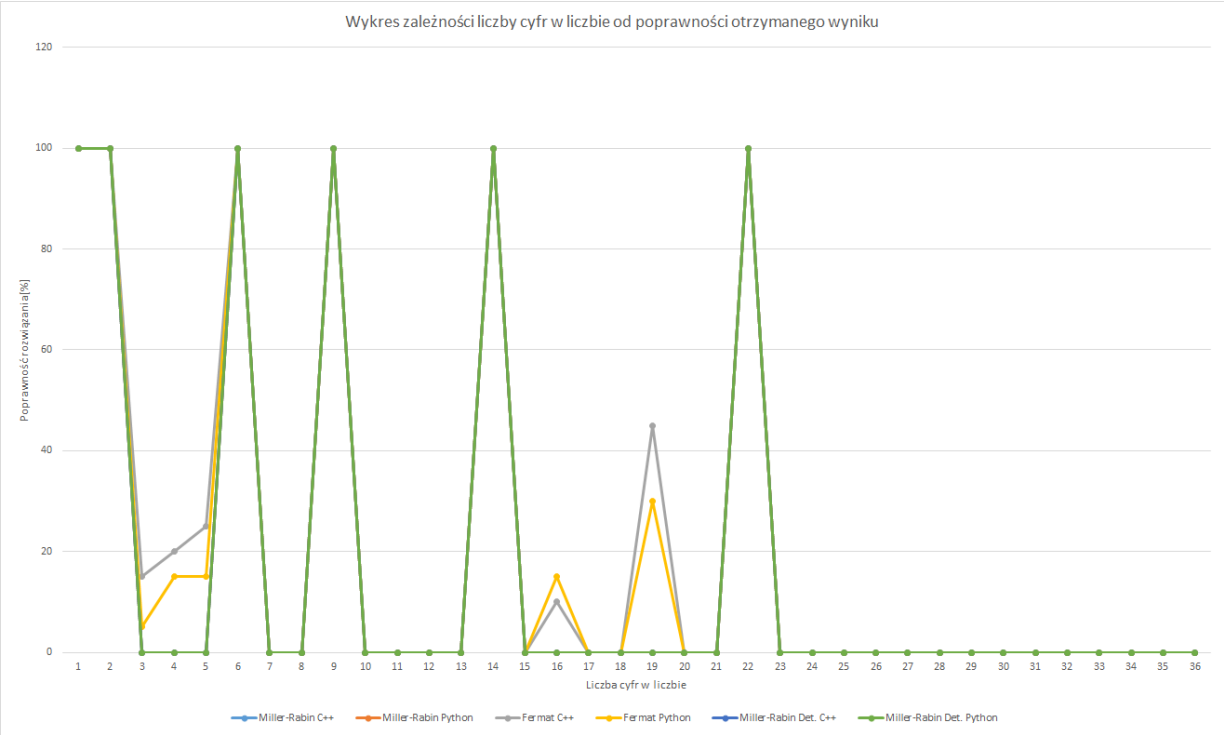
3.2 Analiza jakościowa

W tej sekcji pragnę zająć się analizą wyników otrzymanych przez poszczególne algorytmy. Poniżej prezentuję wykres przedstawiający zależności pomiędzy poszczególnymi algorytmami.



Rysunek 5: Wykres przedstawiający różnice w poprawności działania algorytmów dla różnej liczby powtórzeń pętli wewnętrznych algorytmów.

Na podstawie powyższego wykresu, można wyraźnie zaobserwować, że jedyne błędne rozwiązania generowane są dla algorytmu Fermata w obydwu językach. W związku z tym poniżej prezentuję diagram zależności liczby cyfr w danej liczbie od poprawności wyniku, dla danych w których pętla wewnętrzna algorytmu ustawiona była na pięć powtórzeń, w celu analizy powstawania błędnych wyników.



Rysunek 6: Wykres zależności liczby cyfr w danej liczbie od poprawności wyników dla danych o największych błędach poprawności wyników.

Na podstawie powyższego wykresu zaobserwować możemy w sposób dokładny, które liczby powodują generowanie nieprawidłowych wyników. Zgodnie z informacjami zawartymi w literaturze był to efekt spodziewany, ponieważ błędne wyniki generował algorytm Fermata dla liczb Carmichaela. Również na podstawie dostępnych źródeł potwierdził się fakt, iż nie jest to zjawisko pewne dla każdej sytuacji, a jedynie zwiększanie lub zmniejszanie prawdopodobieństwa wystąpienia błędnego wyniku dla testu pierwszości Fermata. Na podstawie danych znajdujących się na Rysunku nr 5. zaobserwować można, że wzraz ze wzrostem liczby przebiegów wewnętrznej pętli algorytmu, prawododobieństwo wystąpienia błędnego wyniku spada.

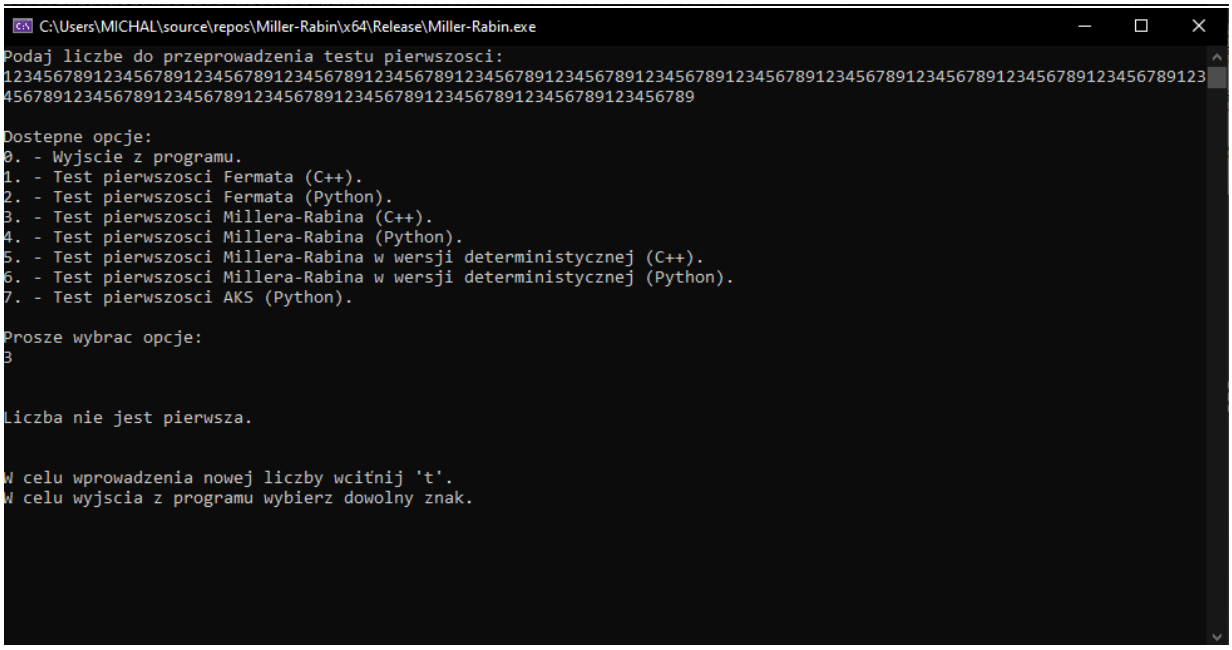
W związku z powyższymi obserwacjami, oraz na podstawie rysunków nr 5 i 6, można przyjąć, iż zaimplementowane algorytmy działają w sposób spodziewany i poprawny.

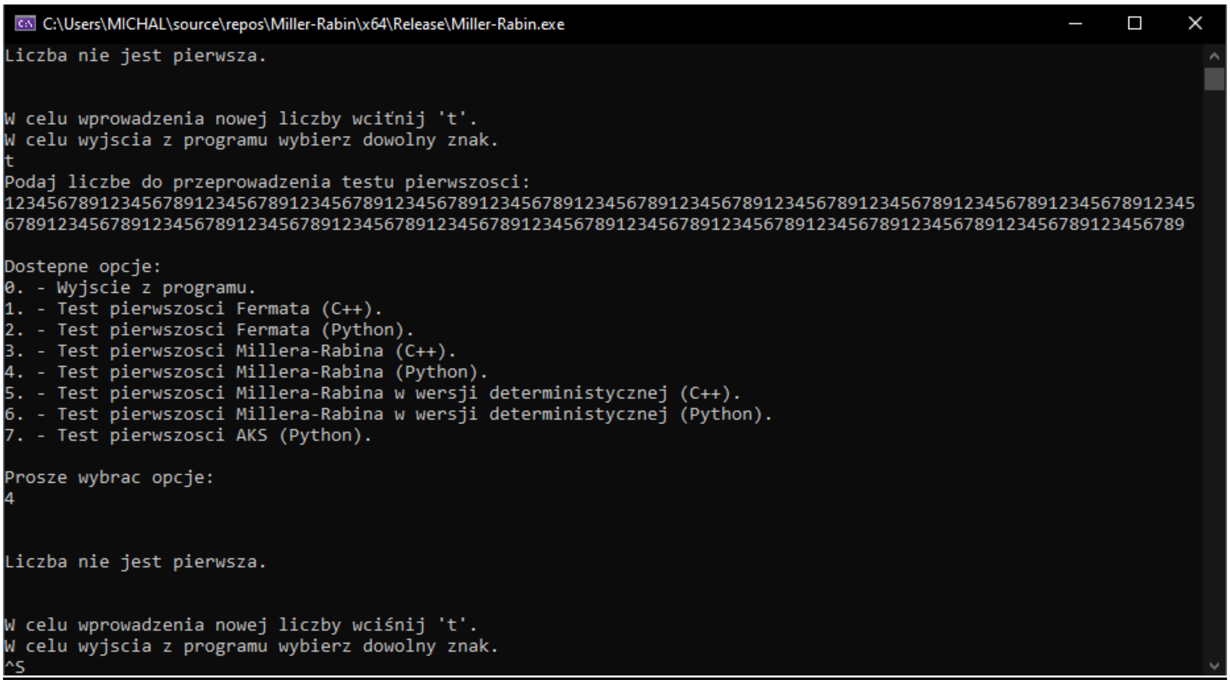
4 Podsumowanie, wnioski i krytyka rozwiązania

Przedstawiona w niniejszej pracy problematyka nie wyczerpuje w całości zagadanienia jakim są testy pierwszości. Istnieje wiele innych wersji testów, które nie zostały przeze mnie przedstawione, jednakże nawiązując do przygotowanego konspektu projektu większość celów została przeze mnie zrealizowana. W porównaniu do początkowych założeń nie został zaimplementowany jedynie naiwny algorytm wyznaczania liczb pierwszych, jednakże w zamian zostały zrealizowane dodatkowe algorytmy w języku C++, których nie uwzględniałem w początkowym projekcie. Uważam, że sporą zaletą było zaimplementowanie struktury do obliczeń na liczbach większych niż standardowe typy danych w języku C++, zamiast wdrażanie gotowej biblioteki do obliczeń na większych liczbach. Dodatkową pozytywną rzeczą jaką udało mi się wdrożyć było wykorzystanie skryptów Pythona wewenątrz projektu pisanego w C++, dzięki czemu umożliwiło mi to porównywnanie napisanych algorytmów w obydwu językach przy użyciu tej samej biblioteki do pomiaru czasów, tak aby otrzymane wyniki były jak najbardziej miarodajne. Ostatnim sukcesem, który pragnę przedstawić jest poznanie odpowiednich algorytmów sprawdzających pierwszość danej liczby, oraz umiejętne ich zaimplementowanie, co potwierdziła chociażby analiza porównawcza algorytmów. Z durgiej strony projekt ten nie jest wolny od wad. Pierwszą z ogromnych wad, jest niepraktyczna implementacja algorytmu AKS do głównego projektu tak, aby móc przetestować go i porównanć z innymi algorytmami. Drugą zdecydowaną wadą jest czas działania algorytmów napisanych w języku C++. Spodziewany czas działania algorytmów miał być znacznie krótszy, niż wyszło to w końcowej wersji programu. Najprawdopodobniej zastosowanie dodatkowych algorytmów, np. algorytmu Karacuby, mogłoby przyspieszyć końcowe działanie algorytmów, jednakże nie było to główne zadanie niniejszego projektu. Dodatkową czynnością, którą można było wykonać to implementacja gotowej biblioteki do działań na liczbach większych niż standardowe typy danych w języku C++ i porównanie wyników działań algorytmów przy wykorzystaniu zewnętrznej biblioteki z dotychczas zaimplementowanymi rozwiązaniami. Ostatnią krytyczną rzeczą, jaką dostrzegam w projekcie jest to, że mogła zostać zaimplementowana większa liczba testów

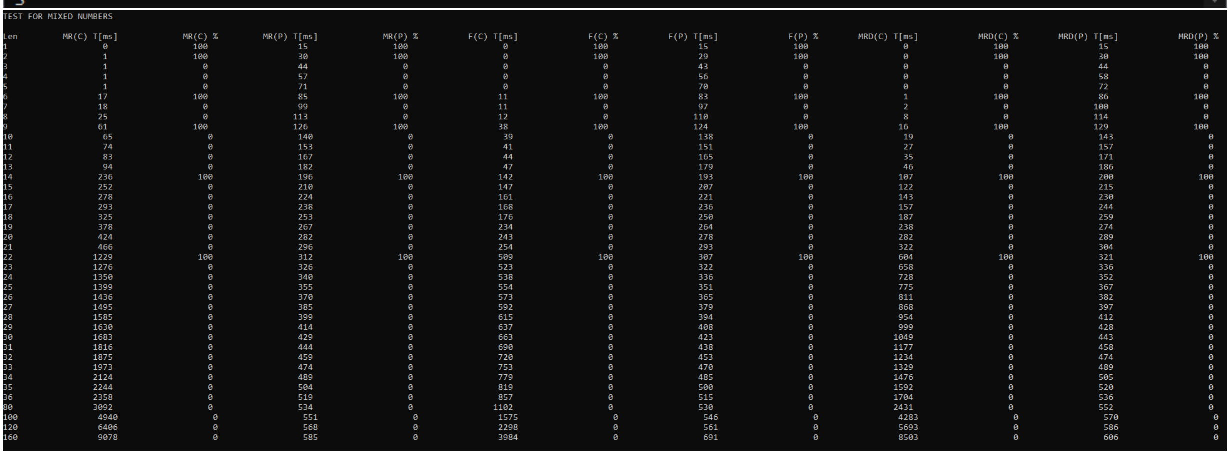
pierwszości. Podsumowując, uważam że zrealizowany projekt spełnia podstawowe założenia, pokrywa się on z początkowym konspektem, a także jest funkcjonalny a zwracane przez niego wyniki są zgodne z oczekiwanymi. Nie jest on wolny od wad i jak większość programów i ten można by usprawnić pod wieloma względami.Na zakończenie pracy, w następnym rozdziale pragnę jedynie przedstawić efekty działania końcowego programu.

5 Zdjęcia z działania programu



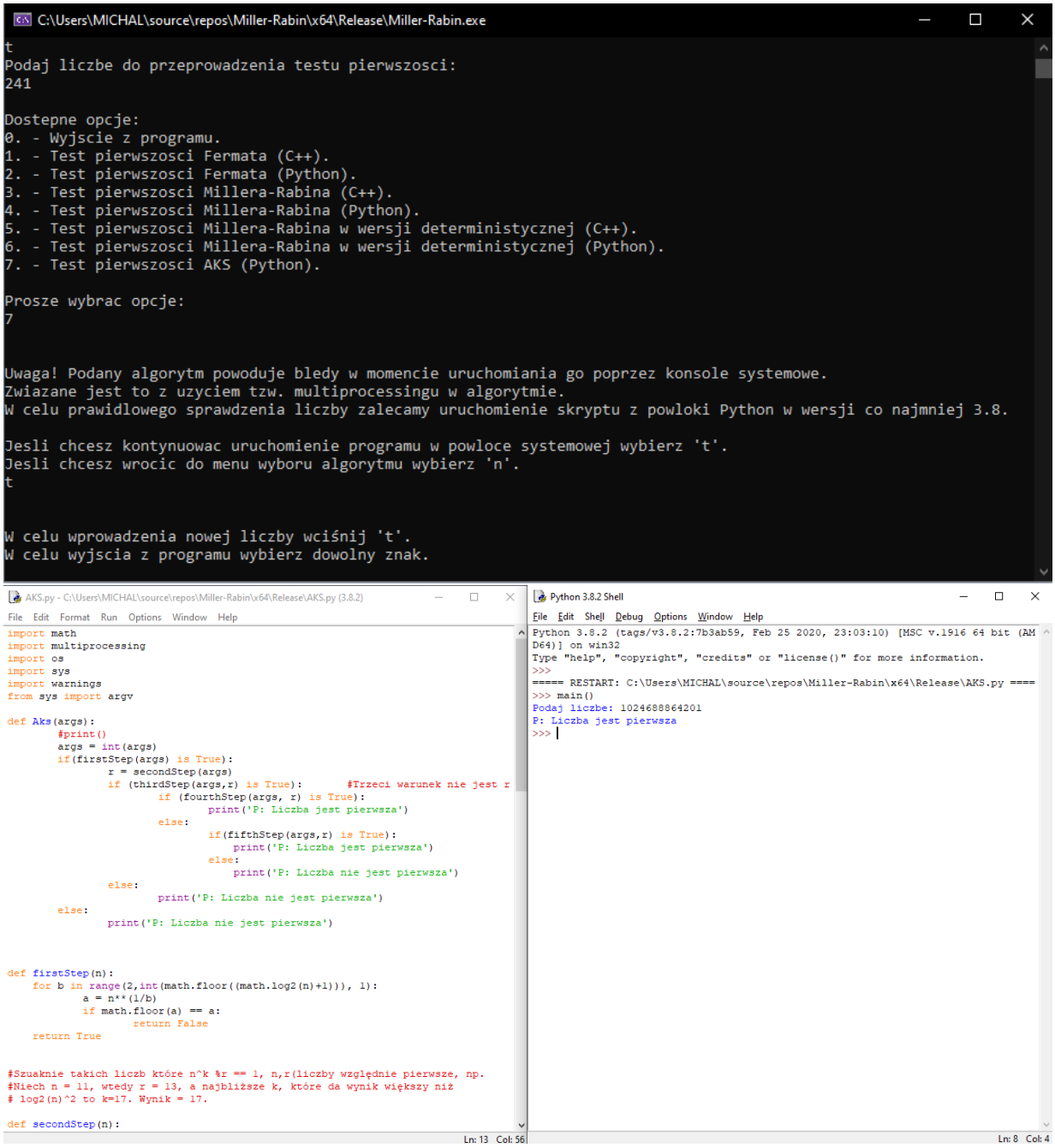


```
C:\Users\MICHAL\source\repos\Miller-Rabin\x64\Release\Miller-Rabin.exe
Liczba nie jest pierwsza.


W celu wprowadzenia nowej liczby wciťnij 't'.
W celu wyjscia z programu wybierz dowolny znak.
t
Podaj liczbe do przeprowadzenia testu pierwszosci:
123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789123456789

Dostepne opcje:
0. - Wyjscie z programu.
1. - Test pierwszosci Fermata (C++).
2. - Test pierwszosci Fermata (Python).
3. - Test pierwszosci Millera-Rabina (C++).
4. - Test pierwszosci Millera-Rabina (Python).
5. - Test pierwszosci Millera-Rabina w wersji deterministycznej (C++).
6. - Test pierwszosci Millera-Rabina w wersji deterministycznej (Python).
7. - Test pierwszosci AKS (Python).

Prosze wybrac opcje:
4


Liczba nie jest pierwsza.


W celu wprowadzenia nowej liczby wciśnij 't'.
W celu wyjscia z programu wybierz dowolny znak.
^S
```



TEST FOR MIXED NUMBERS

Len	MR(C) T[ms]	MR(C) %	MR(P) T[ms]	MR(P) %	F(C) T[ms]	F(C) %	F(P) T[ms]	F(P) %	MRD(C) T[ms]	MRD(C) %	MRD(P) T[ms]	MRD(P) %
1	0	100	15	100	0	100	15	100	0	100	15	100
2	1	100	30	100	0	100	29	100	0	100	30	100
3	1	0	44	0	0	0	43	0	0	0	44	0
4	1	0	57	0	0	0	56	0	0	0	58	0
5	1	0	71	0	0	0	70	0	0	0	72	0
6	17	100	85	100	11	100	83	100	1	100	86	100
7	18	0	99	0	11	0	97	0	2	0	100	0
8	25	0	113	0	12	0	110	0	8	0	114	0
9	61	100	126	100	38	100	124	100	16	100	129	100
10	65	0	140	0	39	0	138	0	19	0	143	0
11	74	0	153	0	41	0	151	0	27	0	157	0
12	83	0	167	0	44	0	165	0	35	0	171	0
13	94	0	182	0	47	0	179	0	46	0	186	0
14	236	100	196	100	142	100	193	100	107	100	200	100
15	252	0	210	0	147	0	207	0	122	0	215	0
16	278	0	224	0	161	0	221	0	143	0	230	0
17	293	0	238	0	168	0	236	0	157	0	244	0
18	325	0	253	0	176	0	250	0	187	0	259	0
19	378	0	267	0	234	0	264	0	238	0	274	0
20	424	0	282	0	243	0	278	0	282	0	289	0
21	466	0	296	0	254	0	293	0	322	0	304	0
22	1229	100	312	100	509	100	307	100	604	100	321	100
23	1276	0	326	0	523	0	322	0	658	0	336	0
24	1350	0	340	0	538	0	336	0	728	0	352	0
25	1399	0	355	0	554	0	351	0	775	0	367	0
26	1436	0	370	0	573	0	365	0	811	0	382	0
27	1495	0	385	0	592	0	379	0	868	0	397	0
28	1585	0	399	0	615	0	394	0	954	0	412	0
29	1630	0	414	0	637	0	408	0	999	0	428	0
30	1683	0	429	0	663	0	423	0	1049	0	443	0
31	1816	0	444	0	690	0	438	0	1177	0	458	0
32	1875	0	459	0	720	0	453	0	1234	0	474	0
33	1973	0	474	0	753	0	470	0	1329	0	489	0
34	2124	0	489	0	779	0	485	0	1476	0	505	0
35	2244	0	504	0	819	0	500	0	1592	0	520	0
36	2358	0	519	0	857	0	515	0	1704	0	536	0
80	3092	0	534	0	1102	0	530	0	2431	0	552	0
100	4940	0	551	0	1575	0	546	0	4283	0	570	0
120	6406	0	568	0	2298	0	561	0	5693	0	586	0
160	9078	0	585	0	3984	0	691	0	8503	0	606	0


C:\Users\MICHAL\source\repos\Miller-Rabin\x64\Release\Miller-Rabin.exe
t
Podaj liczbe do przeprowadzenia testu pierwszosci:
241
Dostepne opcje:
0. - Wyjscie z programu.
1. - Test pierwszosci Fermata (C++).
2. - Test pierwszosci Fermata (Python).
3. - Test pierwszosci Millera-Rabina (C++).
4. - Test pierwszosci Millera-Rabina (Python).
5. - Test pierwszosci Millera-Rabina w wersji deterministycznej (C++).
6. - Test pierwszosci Millera-Rabina w wersji deterministycznej (Python).
7. - Test pierwszosci AKS (Python).
Prosze wybrac opcje:
7
Uwaga! Podany algorytm powoduje bledy w momencie uruchomiania go poprzez konsole systemowe.
Zwiazane jest to z uzyciem tzw. multiprocessingu w algorytmie.
W celu prawidlowego sprawdzenia liczby zalecamy uruchomienie skryptu z powloki Python w wersji co najmniej 3.8.
Jesli chcesz kontynuowac uruchomienie programu w powloce systemowej wybierz 't'.
Jesli chcesz wrocic do menu wyboru algorytmu wybierz 'n'.
t
W celu wprowadzenia nowej liczby wciśnij 't'.
W celu wyjscia z programu wybierz dowolny znak.
AKS.py - C:\Users\MICHAL\source\repos\Miller-Rabin\x64\Release\AKS.py (3.8.2)
File Edit Format Run Options Window Help
import math
import multiprocessing
import os
import sys
import warnings
from sys import argv
def Aks(args):
#print()
args = int(args)
if(firstStep(args) is True):
r = secondStep(args)
if (thirdStep(args,r) is True):
#Trzeci warunek nie jest r
if (fourthStep(args, r) is True):
print('P: Liczba jest pierwsza')
else:
if(fifthStep(args,r) is True):
print('P: Liczba jest pierwsza')
else:
print('P: Liczba nie jest pierwsza')
else:
print('P: Liczba nie jest pierwsza')
else:
print('P: Liczba nie jest pierwsza')
def firstStep(n):
for b in range(2,int(math.floor((math.log2(n)+1))), 1):
a = n**(1/b)
if math.floor(a) == a:
return False
return True
#Szuaknie takich liczb które n^k %r == 1, n,r(liczby względnie pierwsze, np.
#Niech n = 11, wtedy r = 13, a najbliższe k, które da wynik większy niż
# log2(n)^2 to k=17. Wynik = 17.
def secondStep(n):
Ln: 13 Col: 56
Python 3.8.2 Shell
File Edit Shell Debug Options Window Help
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 23:03:10) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\MICHAL\source\repos\Miller-Rabin\x64\Release\AKS.py =====
>>> main()
Podaj liczbe: 1024688864201
P: Liczba jest pierwsza
>>>
Ln: 8 Col: 4

Literatura

[1] Opis testu pierwszości Fermata - http://www.algorytm.org/algorytmy-arytmetyczne/test-pierwszosci-test-fermata.html

[2] Opis małego twierdzenia Fermata - https://eduinf.waw.pl/inf/alg/001_search/0018.php

[3] Opis algorytmu szybkiego potęgowania modularnego - http://smurf.mimuw.edu.pl/node/836

[4] Opis wraz z pseudokodem algorytmu szybkiego potęgowania modularnego - http://www.algorytm.org/algorytmy-arytmetyczne/szybkie-potegowanie-modularne.html

[5] Opis testu Millera-Rabina - http://www.algorytm.org/algorytmy-arytmetyczne/test-pierwszosci-test-millera-rabina.html

[6] Opis testu Millera-Rabina - https://eduinf.waw.pl/inf/alg/001_search/0019.php

[7] Opis poszczególnych wartości wykorzystanych do deterministycznej wersji algorytmu Millera-Rabina - https://primes.utm.edu/prove/prove2_3.html

[8] Opis podstawowych wzorów matematycznych - http://www.matematyka.wroc.pl/book/wzory-skróconego-mnożenia

[9] Opis algorytmu AKS - http://www.aks.bluhost.pl/

[10] Opis algorytmu AKS - https://pl.wikipedia.org/wiki/Test_pierwszo%C5%9Bci_AKS

[11] Spis liczb pierwszych potrzebnych do testowania algorytmów - https://primes.utm.edu/curios/index.php